Electronic Notes in Theoretical Computer Science 36 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume36.html 17 pages

Rewriting Logic as a Framework for Generic Verification Tools

Martin Leucker

Lehrstuhl für Informatik II
Aachen University of Technology
D–52056 Aachen, Germany
leucker@informatik.rwth-aachen.de

Thomas Noll

Department of Teleinformatics
Royal Institute of Technology
S–16440 Kista, Sweden
noll@it.kth.se

Abstract

In this paper we propose to employ Rewriting Logic as a generic and uniform approach to support different specification languages for distributed systems in verification tools. We present a compiler generator which, given the definition of a language, automatically generates a corresponding model–checking tool. More specifically, the syntax and semantics of the specification language has to be described in terms of Rewriting Logic, a unified semantic framework for concurrency. From this definition a compiler is derived which is capable of parsing a concrete system specification and of computing the corresponding semantic object, such as a labeled transition system. The compiler is linked together with the existing verification platform TRUTH to obtain a model–checking tool for the specification language in question. As an example we formulate Milner's CCS, and we conclude by describing the practical results obtained so far and by presenting directions for future work.

1 Introduction

Formal methods are becoming more and more popular for the specification and verification of distributed systems. Numerous case studies have shown that mathematical methods can help to find errors in the design of complex hardware and software systems (see [4] for an overview). A formal specification helps to understand the system under development. Furthermore, a common and precise basis for reasoning about the system is given. Rigorous techniques are also gaining commercial success, e.g. companies such as Intel, National

Semiconductor or Texas Instruments are establishing new departments for formal methods (see for example the job adverts in [9]).

The application of formal methods requires the availability of supporting tools because formal methods are especially adequate for the design of large systems where an *ad hoc* or conventional software engineering approach is not reasonable. Generally speaking, large systems consist of distributed processes working together concurrently. While the distribution of the processes usually does not involve any conceptual problems, the concurrent behavior makes the system difficult to understand. Therefore, we put our emphasis on analyzing concurrent systems. During the last years several prototypes of corresponding tools have been developed, e.g. CWB ([25]), NCSU–CWB ([8]), SPIN ([12]), and the symbolic model checker SMV ([18]). Most of the tools are tailored to a specific syntactic and semantic setting, e.g. CCS with transition–system semantics and μ–calculus model checking, like in the case of CWB.

Our goal is to support the employment of new specification formalisms and semantic domains by a compiler generator which, given the definition of a specification language, automatically generates a corresponding model–checking tool.

Rewriting Logic was proposed in [19,20] as a unified semantic framework for concurrency. In this approach the state of a system is represented by an equationally–defined equivalence class of terms, and transitions correspond to rewriting operations on the representatives. Hence Rewriting Logic supports both the definition of specification formalisms and, by employing (equational) term rewriting methods, the execution or simulation of concrete system specifications. This *executable specification* property is exploited by our SLC specification language compiler generator which, given a Rewriting Logic description of the syntax and semantics of the specification language in question, derives a compiler which parses any given system specification and computes the associated semantic object. SLC is part of our Truth verification tool ([15]) which can subsequently be used to visualize and to analyze the semantics. In particular it can be employed to perform model checking, i.e. to verify that the system under consideration fulfills certain conditions specified as formulae of some formal logic.

In [7] a process algebra compiler was presented with a similar motivation. Given a process algebra description, the compiler generates a frontend for the NCSU–CWB. The description consists of a specification for the syntax and for the semantics. To define the latter, structural operational semantics (SOS) rules must be provided. Hence, this approach is just suitable for formalisms that have a (natural) SOS–style semantics which does not involve any equational reasoning. We believe that our approach is more powerful since it supports the structural description of states via equations, which leads to small models for the underlying specifications. Furthermore, numerous publications have shown that most of the existing specification formalisms can be expressed via Rewriting Logic in a very elegant manner (see [21] for an

overview).

With regard to the implementation of our SLC compiler generator, one might think of employing one of the existing Rewriting Logic interpreters (such as Maude or CafeOBJ [5,10]) or some general (equational) term rewriting tool (like ELAN [3]). However, neither of these turned out to be well suited for our purposes.

Maude is a fast and powerful Rewriting Logic interpreter which is highly optimized. Therefore, the current implementation only supports deterministic rewriting. Because of the reflection properties of Rewriting Logic, this is no limitation in general. However, Rewriting Logic descriptions of specification languages tend to become much more concise if nondeterminism is directly supported. A description of CCS, for example, within our tool consists of about 200 lines (cf. Section 3) while a comparable implementation in Maude comprises 750 lines ([27]). Furthermore, our description is very similar to the presentation given in the literature ([23]) while the version for Maude had to be designed employing reflections and strategies.

ELAN is a powerful general–purpose conditional rewriting tool. However, our attempt to employ it as the underlying rewriting engine for our verification tool failed. The computation of the successors of the current system state is one of the basic steps in the state–space analysis which has to be carried out efficiently. Therefore, a compiling approach as well as a fast interface for accessing the compiled code are necessary. At the time of developing the SLC the ELAN compiler turned out to be unstable while the ELAN interpreter was rather slow.

The most serious handicap for employing a foreign rewrite engine within our SLC is the interface problem. Using a string–based interface for accessing a stand–alone program is much too slow. Hence, to be applicable, a rewrite engine must be provided in form of a library which can be linked to the final system. Furthermore, the same data structures should be employed to avoid expensive marshaling.

We therefore decided to develop our own generator which, given a Rewriting Logic definition of the specification formalism, derives program code meeting Truth's interfaces. All together is compiled to obtain the final verification tool for the desired formalism. Thus the user of the final tool need not have any experience with Rewriting Logic at all. He or she just has to give the concrete system specifications according to the specification formalism.

Of course one cannot expect to obtain highly–efficient verification tools using this fairly general approach. Instead, it should be considered as a kind of prototype generator in which new and modified specification formalisms and their semantics can be easily studied and tested ([16]). Here our compiler–generating approach turns out to be very useful in supporting the incremental design of the syntax and semantics of a specification language. Once an appropriate representation has been determined, the generated code can be further optimized by hand.

In the remainder of this paper we introduce our approach in greater detail. Section 2 gives an overview of Truth, the underlying verification tool. In Section 3 we sketch the formal basis of the compiler generator SLC, the Rewriting Logic formalism, and its application as a modeling framework for specification formalisms. Section 4 describes the implementation of the compiler generator. In Section 5 we comment on the results obtained so far. Finally, in Section 6, we draw some conclusions and describe possible future work .

2 Truth

Truth, the tool underlying our SLC specification language compiler, is a software platform for the verification of concurrent systems ([15]). It was developed to serve as a kind of prototype testbed which supports the integration and testing of new formalisms and methods in this area and to be used in the education of formal methods. [1]

In Truth concurrent systems are specified in CCS, a well–known process algebra ([23]). From the specification a labeled transition system is built. Its desired properties can be expressed using the μ–calculus ([14]), a powerful logic which allows to describe various safety, liveness, and fairness properties ([11]). It semantically subsumes the temporal logics CTL (whose operators are implemented as macros in Truth), CTL*, and LTL. The tableau–based model checker proposed in [6] is used to test whether a formula is satisfied by the transition system. Furthermore, a local game–based model–checking algorithm has been integrated into the system supporting interactive debugging of the specification.

With respect to functionality, runtime behavior, and memory usage, Truth is comparable to the Concurrency Workbench ([25]). However, an additional feature of Truth is the interactive, graphical, and process–oriented visualization of CCS processes.

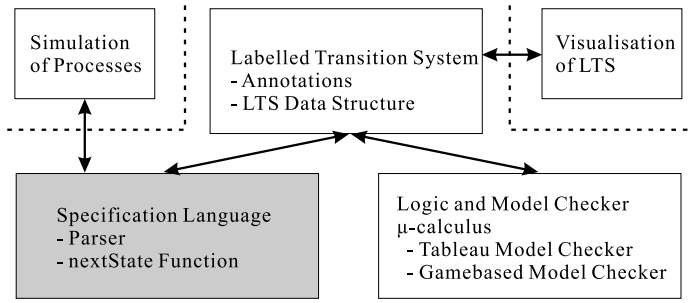

Fig. 1. The Structure of Truth

Truth is implemented in Haskell, a general–purpose, fully functional programming language. We have chosen a modular design that facilitates

[1] see URL `http://www-i2.informatik.rwth-aachen.de/Research/MCS/Truth/`

modifications of the system (see Figure 1). Especially the small and simple interface of the specification language module supports the integration of other specification formalisms. The specification language compiler simplifies this task even more since it automatically generates the Haskell code corresponding to the given specification language definition, and substitutes it for the current CCS implementation. Informally said, the program code produced by SLC substitutes the code depicted as the shadowed box in Figure 1.

3 The SLC Specification Language Compiler Generator

In this section we present an extension of Truth which facilitates the use and investigation of new specification formalisms and semantic domains. SLC is a compiler generator which, given the definition of a (high–level) specification language, automatically generates a corresponding Truth frontend. Figure 2 illustrates the working principle. Note that the shadowed box corresponds to the one in Figure 1.

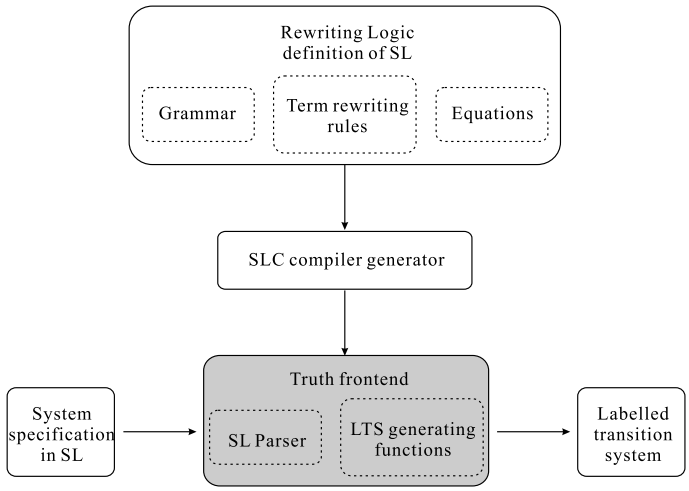

Fig. 2. Application of SLC to a specification language SL

The syntax and semantics of the specification language under consideration has to be determined using a variant of the *Rewriting Logic* formalism, which aims at a separate description of the static and the dynamic aspects of a distributed system ([19,20]). More exactly, it distinguishes the laws describing the structure of the system's states from the rules which specify its possible transitions. The two aspects are respectively formalized as a set of equations and as a (conditional) rewriting system. Both structures operate on states, represented as (equivalence classes of) expressions of the specification language under consideration.

Aiming at its practical usability, we propose some fundamental restrictions of the Rewriting Logic framework. We will motivate them by first presenting a definition of the well–known process algebra CCS, the Calculus of Commu-

nicating Systems (see [23]), and by pointing out the problems which arise with the original Rewriting Logic approach. The exposition is inspired by [28].

Example 3.1 *The Rewriting Logic definition* $\mathfrak{C} = (\Sigma, E, R)$ *is given by the following components, using the set* $Var = \{x, x', y, y', z, \alpha, p\}$ *of variables:*

- $\Sigma = \{\mathit{nil}^{(0)}, .^{(2)}, |^{(2)}, +^{(2)}, {}^{-(1)}, \tau^{(0)}, \{\ \}^{(2)}\} \cup A \cup I$ *is a* signature[2] *containing the CCS operators with their associated arities (restrictions and relabelings are omitted for simplification) and, additionally, a set of actions* $A = \{a^{(0)}, b^{(0)}, \ldots\}$ *and a set of process identifiers* $I = \{P^{(0)}, Q^{(0)}, \ldots\}$.
- $E = \{x + (y + z) = (x + y) + z,\ x + y = y + x,\ x + \mathit{nil} = x,$
$$x \mid (y \mid z) = (x \mid y) \mid z, \quad x \mid y = y \mid x, \quad x \mid \mathit{nil} = x,\ \bar{\bar{\alpha}} = \alpha\}$$
is a set of equations *which define both the parallel operator* $\mid$ *and the choice operator* $+$ *to be associative and commutative, which declare the empty process* nil *as the neutral element of both operations, and which state an idempotence property of the complement operator* $\bar{}$.
- $R = \{$ $\quad (PRE)\dfrac{}{[\alpha.x] \longrightarrow [\{\alpha\}x]} \qquad (SUM)\dfrac{[x] \longrightarrow [\{\alpha\}x']}{[x + y] \longrightarrow [\{\alpha\}x']}$

$$(PAR)\frac{[x] \longrightarrow [\{\alpha\}x']}{[x \mid y] \longrightarrow [\{\alpha\}(x' \mid y)]} \qquad (COM)\frac{[x] \longrightarrow [\{\alpha\}x'] \quad [y] \longrightarrow [\{\bar{\alpha}\}y']}{[x \mid y] \longrightarrow [\{\tau\}(x' \mid y')]}$$

$$(DEF)\frac{p = x \quad [x] \longrightarrow [\{\alpha\}x']}{[p] \longrightarrow [\{\alpha\}x']}\}$$

is a set of conditional term rewriting rules *describing the operational semantics of prefixing (PRE), choice (SUM), of the parallel product without (PAR) and with (COM) communication, and of the process identifier definition (DEF). Here the additional binary operator* $\{\ \}$ *is used to simulate the (action) labels of the transition steps, which are not provided in the formal definition of rewrite rules. Note that the symmetric variants of (SUM) and (PAR) can be omitted because of the commutativity of* $+$ *and* $\mid$*, respectively.*

The problems which we encountered with such a definition are the following:

(i) (Conditional) term rewriting systems usually induce *congruence relations*, that is, a rule can be applied in arbitrary contexts provided that the left–hand side matches the respective subterm and that the conditions are fulfilled. In the case of CCS this means that the expression $a.b.\mathsf{nil}$ can evolve to $a.\{b\}\mathsf{nil}$, which should clearly be forbidden since this implies

[2] Actually the signature is *typed*, distinguishing e.g. actions from processes (see also Example 3.2).

that the b step occurs before the a step:

$$\text{(Congruence)}\frac{\text{(PRE)}\ \dfrac{}{[b.\mathsf{nil}] \longrightarrow [\{b\}\mathsf{nil}]}}{[a.b.\mathsf{nil}] \longrightarrow [a.\{b\}\mathsf{nil}]}$$

(ii) Since (conditional) term rewriting modulo equational theories is generally too complex or even undecidable, it is not possible to admit arbitrary equations.

(iii) In a conditional rewriting rule $\rho \in R$, which is of the general form

$$(\rho)\frac{c_1 \longrightarrow d_1 \quad \ldots \quad c_k \longrightarrow d_k}{l \longrightarrow r}$$

where $k \geq 0$, and where the single terms are built up from operators from Σ and variables from Var, the use of variables must be restricted in such a way that the "data flow" between them can be implemented deterministically. For example, if c_1 would contain a variable that is not used in l, it would be unclear how to evaluate it when applying ρ to some instance of l.

We therefore suggest the following modifications:

(i) One solution to this problem is known as *order–sorted rewriting*. The idea is to introduce a hierarchical type system on terms and to employ rewriting rules only if the resulting terms are typable ([17]). (In the above example, one would consider $a.\{b\}\mathsf{nil}$ being untypable.) However, this in general demands for runtime types and runtime type checks within the resulting tool. Since our interest in Rewriting Logic is motivated by the wish to implement efficient verification algorithms for distributed systems, the overhead which is undoubtly caused by this extension is too expensive.

We therefore propose to distinguish *compatible* operators from *incompatible* ones, and to allow the transition rules in R to be applied in "compatible contexts" only. More exactly, a rewriting step at some position of a term is possible only if every symbol on the path from the root to this position is compatible. Formally this is reflected by the following rule, valid for every compatible symbol f:

$$\text{(CMP)}\frac{s_1 \longrightarrow t_1 \quad \ldots \quad s_k \longrightarrow t_k}{f(s_1, \ldots, s_k) \longrightarrow f(t_1, \ldots, t_k)}$$

It is obvious that the $\{\ \}$ symbol which is used to simulate action labels (see Example 3.1) is the only compatible operator in the CCS specification. The compatibility of the prefix operator, for example, would allow for the incorrect behavior above. Similar arguments apply to the remaining operators.

(ii) Following the ideas of Viry in [28,29], we propose to decompose E into a set of directed equations ER (that is, a term rewriting system) and into a set AC expressing associativity and commutativity of certain binary operators in Σ. In the case of CCS, one would e.g. choose $ER = \{x + \mathsf{nil} \longrightarrow x, x \mid \mathsf{nil} \longrightarrow x, \bar{\bar{\alpha}} \longrightarrow \alpha\}$ and $AC = \{x + (y + z) = (x + y) + z, x + y = y + x, x \mid (y \mid z) = (x \mid y) \mid z, x \mid y = y \mid x\}$. Given that ER is terminating modulo AC, then rewriting by R modulo E can be implemented by a combination of normalizing by ER and rewriting by R, both modulo AC.

From a semantical point of view this means that we are working with Abelian monoids or, in other words, multisets, which turn out to be appropriate for many applications in the area of concurrency and distributed systems. In the next section we will address the question under which premises this restriction of the fully–equational semantics is sound and complete.

(iii) In a rewrite rule $\rho \in R$ of the above form, we require that, for every $i \in \{1, \ldots, k\}$,

$$Var(c_i) \subseteq Var(l) \cup \bigcup_{j=1}^{i-1} Var(d_j)$$

and

$$Var(r) \subseteq Var(l) \cup \bigcup_{i=1}^{k} Var(d_i).$$

In the next section we will see that, under these assumptions, every rewriting step can be computed by evaluating the rules in a "depth–first left–to–right" fashion.[3] Note that this requirement is fulfilled by the CCS specification given in Example 3.1.

The following example shows the essential parts of a corresponding definition for CCS using the SLC syntax, which follows the input format of the NCSU Process Algebra Compiler (cf. [7]).

Example 3.2 *The definition starts with a declaration of the operators with their associated types, their textual output representation, and their associativity and commutativity (`AC`) properties.*

```
ALGEBRA CCS

sorts
  definition, exp, act

cons
  Definition : string * exp -> definition ("_ = _")
  Id    : string -> exp                   ("_")
```

[3] This syntactic restriction corresponds to the 3–CTRS property of ordinary conditional term rewriting systems (cf. [22]).

```
  Nil   : unit -> exp                    ("nil")
  Pref  : act * exp -> exp               ("_._")
  Plus  : exp * exp -> exp               ("_ + _")   (AC)
  Par   : exp * exp -> exp               ("_ | _")   (AC)
  ...
```

*The next part describes the syntactic structure of the specifications. It declares the tokens and their associativity (**left** or **right**) and priority (where higher numbers denote a higher priority), the nonterminal symbols, and the context–free rules together with their abstract syntax tree representation.*

```
SYNTAX

tokens
  "[A-Z][A-Z0-9]*" => PROCID of String
  "="   => EQUAL
  "nil" => NIL
  "."   => DOT
  "\+"  => PLUS
  "\|"  => PAR
  ...

priorities
  left 10 PLUS
  left 20 PAR
  right 90 DOT
  ...

nonterminals
  def of definition   (SYSTEM)
  exp of exp          (STATE)
  ...

grammar
  def   : PROCID EQUAL exp (Definition(PROCID, exp))
  exp   : PROCID           (Id(PROCID))
   | NIL                   (Nil())
   | act DOT exp           (Pref(act, exp))
   | exp PLUS exp          (Plus(exp1, exp2))
   | exp PAR exp           (Par(exp1, exp2))
  ...
```

Finally, the SEMANTIC part gives the oriented equations in ER and the R rules which define the transitional behavior of the terms. Here explicit transition labels can be used. They are automatically encoded in term structures as shown in Example 3.1.

```
SEMANTICS
```

```
vars
  a : act
  x, y, x', y' : exp
  ...

equations
  x | nil -> x
  ...

rules

(PRE)-------------
      a.x -(a)-> x

        x -(a)-> x'
(SUM)----------------
      x + y -(a)-> x'

          x -(a)-> x'
(PAR)---------------------
      x | y -(a)-> x' | y

      x -(a)-> x', y -('a)-> y'
(COM)--------------------------
       x | y -(tau)-> x' | y'

      p = x in Set, x -(a)-> x'
(DEF)-------------------------
      Id(p) -(a)-> x'
...

end
```

4 Implementation of SLC

As we explained in the previous section, SLC is a compiler generator which, given the definition of a specification language, automatically generates a corresponding Truth frontend which can be used to read and to evaluate expressions of the respective specification language. To be more specific, from every part of a language definition like the one in Example 3.2 certain components of the Truth system are derived:

- The `ALGEBRA` part induces a set of constructor symbols Σ which is used to internally represent the expressions of the specification language under consideration. Moreover it determines which symbols have to be considered in the AC–matching test, which is based on the algorithms described in [13], and which symbols are compatible with the transition relation.

- The `SYNTAX` part yields an expression parser which performs two tasks. First, it is used by SLC itself to analyze the syntactic structure of the terms occurring in the `SEMANTICS` part. Second, it is employed as the parser in the Truth frontend.
- Finally, the oriented equations and the transition rules in the `SEMANTICS` part supply the basis for those functions which implement the ER reduction and the R transition relation, respectively.

In our implementation of the transition relation, the current state of the system is represented by a Σ–term which is in normal form with respect to the ER rules. The central point is the combination of normalization by ER and of rewriting by R, both modulo AC, which avoids the inherent difficulties of term rewriting modulo arbitrary equational theories. To illustrate the working principle of our implementation, we again consider a transition rule $\rho \in R$ of the general form

$$(\rho)\frac{c_1 \longrightarrow d_1 \quad \ldots \quad c_k \longrightarrow d_k}{l \longrightarrow r}$$

where $k \geq 0$ and where l, r, c_i, d_i are Σ–terms over some set Var of variables.

If ER is terminating modulo AC, then the following algorithm can be used to determine the next possible states of a system whose current state is given by a term s in ER–normal form:

(i) For every subterm s' of s which is reachable by a path labeled with compatible symbols only, test whether s' is AC–equivalent to some instance $l\sigma$ of the left–hand side l of a rule $\rho \in R$ of the above form. [4]

(ii) For every condition $c_i \longrightarrow d_i$, compute ER–normal forms c' and d' of $c_i\sigma$ and $d_i\sigma$, respectively (both modulo AC), using standard techniques from equational term rewriting (see e.g. [1]).

(iii) Test recursively whether there exists a transition from c' to d'. This may cause the instantiation of variables which occur in d_i but not in l nor in c_i, leading to a corresponding extension of σ.

(iv) If every condition has been successfully tested, then every ER–normal form of $r\sigma$ is a successor state.

Now we are concerned with the question whether this implementation respects the fully–equational semantics of the transition rules in R, which is defined by allowing the rules in ER to be applied in both directions. The correctness is trivial: since the implementation is obtained just by orienting some of the equations in E, every transition which is computed by the implementation is also contained in the semantics.

[4] Here it is important that the AC–unification problem is decidable and finitary; that is, one can always decide whether two given terms s, t are AC–unifiable and, if they are, determine a finite minimal complete set of substitutions σ such that $s\sigma$ and $t\sigma$ are AC–equivalent (see [2] for details).

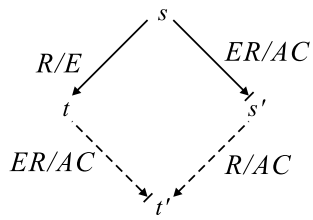

Fig. 3. Coherence property

As usual, the completeness is the hard part. Here we have to ensure that the orientation of the non–AC equations does not restrict the potential successor states. More specifically, if from a state s a successor t is reachable in the fully–equational semantics on the one hand and, on the other hand, s' is an ER–normal form of s modulo AC, then s' should possess an implementational successor t' which is an ER–normal form of t. Figure 3 illustrates this *coherence property*. It is defined and investigated in [29] for the situation that R is a set of *unconditional* rewrite rules. In [26] it is shown that in the conditional case coherence can be reduced to deciding whether finitely many conditional critical pairs of terms are joinable using rules from R and ER.

It can be shown that the CCS definition in Example 3.2 is coherent. For example, the (PAR) rule in R and the rule $\eta : y \mid \mathsf{nil} \longrightarrow y \in ER$ yield, under the condition $x \xrightarrow{\alpha} x'$ and under the substitution $\sigma = [y \mapsto x]$, the *critical peak* shown on the left–hand side in Figure 4, which can be closed as shown on the right–hand side.

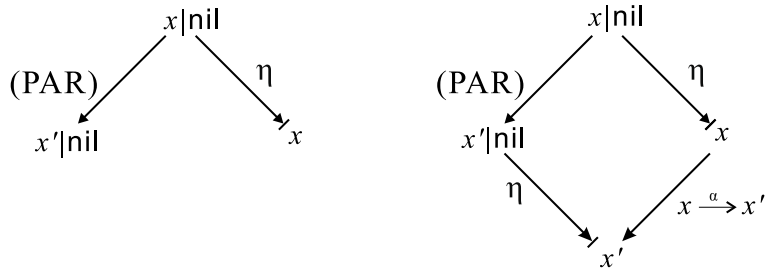

Fig. 4. A critical peak

A prototype of SLC has been implemented in the functional language Haskell. We are currently developing a decision algorithm for the coherence problem, which will also be included in future releases of the SLC compiler generator.

5 Results of the Implementation

We have tested the prototype implementation of our SLC compiler generator using definitions of two important models of concurrency, CCS and Petri nets.

For different system specifications, we have measured both the size of the resulting transition system and the time which is needed to compute it.

It turned out that our compiler–generating approach supports the incremental development of specification language definitions very well. It is easy to modify an existing definition in order to correct errors, or to change the semantics. So SLC can be used as a rapid prototyping tool for language implementations (see also [16]).

One of the main motivations for introducing equations between terms was the expected reduction of the state space. Comparing our Rewriting Logic definition of CCS (see Example 3.2) with the interleaving semantics as implemented by the original Truth tool (cf. Section 2), we have experienced that its benefits depend on the type of system which is to be modeled. In the case of a simple communication protocol like the Alternating Bit Protocol, no reduction could be achieved at all. This is due to the fact that the structure of such a system is static in the sense that it consists of a fixed set of processes (usually a sender, a receiver, and some communication medium) whose positions within the current state do not change. In more dynamic environments such as client–server systems, where new processes are created and where others get stuck in a nil state, considerable reductions can be achieved.

Even more interesting, equations support a novel style of designing systems. Short CCS specifications of dynamic data structures like stacks or queues often involve infinite–state systems, usually due to nil processes which remain when an element is retrieved. This holds even if only a bounded portion of the structure is being used, such as in embedded controller systems. To represent such a system in CCS, the user has to give a (parametric) specification for every possible size of the data structure. Due to the reduction rules in our approach, it is now possible to use the general specification, provided that in the concrete application the number of elements to be stored is bounded. This allows for the development of a generic library supporting common data structures.

Furthermore it is possible to employ an extension of CCS proposed in [23], called *well–terminating processes*. Here a sequential composition operator is defined which can be reduced to the basic CCS operators. However, the use of sequential composition within a recursive process definition yields an infinite–state system in the original semantics. In our setting supporting equations, these systems are automatically reduced to bisimilar finite–state systems.

The positive experiences regarding the memory consumption are somewhat spoiled by the runtimes. It turned out that the computation of the transition system in the (fully equational) SLC–based implementation of CCS is up to one order of magnitude slower than in the original Truth system. We identified the following reasons for this behavior.

Due to the generality of the Rewriting Logic approach, no specific knowledge about the rewriting rules (such as determinacy) can be exploited, which is essential for an efficient implementation. However, we are convinced that this

problem can be solved by employing hashing techniques. We have analyzed the rewriting functions and have detected that often common subgoals are proven multiple times in different rewriting steps. So we applied a memorization method for subgoals that have been considered already. More exactly, during the computation of the transition system every pair consisting of a state and its list of successors is stored within a predefined cache of a fixed size. Given a state we first check (employing hashing) whether the successors have already been computed, avoiding recomputation of the expensive goals in this case. This accelerated the computation for typical examples by a factor of about 10. Further optimization techniques (like update–in–place) which already proved to be valuable in the original Truth implementation of CCS will improve the runtime performance even further.

Another efficiency problem is caused by the AC–unification which is employed several times in every rewriting step. The runtime of the AC–unification algorithm is (potentially) exponential in the size of the terms. Furthermore in the current prototype version of SLC a generic algorithm is employed which represents terms as strings, leading to additional data conversions. Improving the SLC by generating an AC–unification function which works with the internal data structures will avoid this overhead.

6 Conclusion

In this paper we described the design and an implementation of a specification language compiler generator called SLC. Given a specification language definition by its syntax and operational semantics in terms of Rewriting Logic, it generates parsing and semantic functions which can be integrated in tools such as Truth to obtain a verification tool tailored for the respective specification formalism.

We have tested our prototype implementation extensively with CCS. We showed that with our approach considerable reductions of the state space can be achieved. This is due to the possibility to provide equational laws for the system states. Since the turnaround time for the specification language compiler is short, variations of CCS using new process operators or modified (bisimilar) semantics can be handled without spending a lot of effort.

As another example with a different flavor we developed a Rewriting Logic definition for Petri nets. Employing compatible operators and AC–equations, it was easy to define a truly concurrent semantics.

At the moment, we use our tool for describing the π–calculus ([24]) using various semantics. The goal is to obtain a set of operators and a semantics with a minimum of effort for describing advanced telecommunication protocols in a version of the π–calculus.

The main problem of the current implementation of SLC regarding its practical usability is the lacking runtime efficiency of the generated code. However, the prototype was built to gain experiences with our approach. Integrat-

ing standard optimization techniques into the generated code will certainly yield verification tools with a higher degree of efficiency.

References

[1] Baader, F. and T. Nipkow, "Term Rewriting and All That," Cambridge University Press, New York, 1998.

[2] Baader, F. and J. Siekmann, *Unification theory*in: D. Gabbay, C. Hogger and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, Oxford, UK, 1994 .

[3] Borovansky, P., C. Kirchner, H. Kirchner, P. Moreau and M. Vittek, *Elan: A logical framework based on computational systems*, in: *Proc. of the First Int. Workshop on Rewriting Logic*, Electronic Notes in Theoretical Computer Science **4** (1996).
URL `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`

[4] Clarke, E. M. and J. Wing, *Formal methods: State of the art and future directions*, Technical Report CMU-CS-96-178, Carnegie Mellon University (CMU) (1996).
URL `ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-178.ps`

[5] Clavel, M., S. Eker, P. Lincoln and J. Meseguer, *Principles of Maude*, in: J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, Electronic Notes in Theoretical Computer Science **4**, Elsevier, 1996, pp. 65–89.
URL `http://www.csl.sri.com/~clavel/pubs/rwl96b.ps`

[6] Cleaveland, R., *Tableau-based model checking in the propositional mu-calculus*, Acta Informatica **27** (1990), pp. 725–748.
URL `http://www.cs.sunysb.edu/~rance/publications/papers/ai90.ps.gz`

[7] Cleaveland, R., E. Madelaine and S. Sims, *A front-end generator for verification tools*, in: *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Lecture Notes in Computer Science **1019**, 1995, pp. 153–173.
URL `http://www.csc.ncsu.edu/eos/users/s/stsims/WWW/papers/pac.ps`

[8] Cleaveland, R. and S. Sims, *The NCSU concurrency workbench*, in: *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV'96)*, Lecture Notes in Computer Science **1102**, 1996, pp. 394–397.

[9] *The concurrency mailing list.*
URL `http://www.cwi.nl/~bertl/concurrency/concurrency.html`

[10] Diaconescu, R. and K. Futatsugi, "CafeOBJ Report," World Scientific, Singapore, 1998.

[11] Emerson, E. A., "Automated Temporal Reasoning about Reactive Systems," Lecture Notes in Computer Science **1043**, Springer-Verlag Inc., New York, NY, USA, 1996 pp. 41–101.

[12] Grégoire, J.-C., G. J. Holzmann and D. A. Peled, editors, "The Spin Verification System," DIMACS series **32**, American Mathematical Society, 1997, iSBN 0-8218-0680-7, 203p.

[13] Herold, A. and J. J. Siekmann, *Unification in Abelian semigroups*, Journal of Automated Reasoning **3** (1987), pp. 247–284.

[14] Kozen, D., *Results on the propositional mu-calculus*, Theoretical Computer Science **27** (1983), pp. 333–354.

[15] Lange, M., M. Leucker, T. Noll and S. Tobies, *Truth – a verification platform for concurrent systems*in: *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, Springer-Verlag Wien New York, 1999 .

[16] Leucker, M. and T. Noll, *Rapid prototyping of specification language implementations*, in: *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping* (1999), pp. 60–65.

[17] Martí-Oliet, N. and J. Meseguer, *Rewriting logic as a logical and semantic framework*, in: *First International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science **4** (1996), pp. 352–358.

[18] McMillan, K. L., *The SMV system, symbolic model checking - an approach*, Technical Report CMU-CS-92-131, Carnegie Mellon University (1992).

[19] Meseguer, J., *Rewriting as a unified model of concurrency*, in: *Proceedings Concur'90 Conference*, Lecture Notes in Computer Science, Volume 458 (1990), pp. 384–400, also, Report SRI-CSL-90-02R, Computer Science Lab, SRI International.

[20] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[21] Meseguer, J., *Rewriting logic as a semantic framework for concurrency: a progress report*, in: *Seventh International Conference on Concurrency Theory (CONCUR '96)*, Lecture Notes in Computer Science **1119** (1996), pp. 331–372.

[22] Middeldorp, A. and E. Hamoen, *Completeness results for basic narrowing*, Journal of Applicable Algebra in Engineering, Communication and Computing **5** (1994), pp. 313–353.

[23] Milner, R., "Communication and Concurrency," International Series in Computer Science, Prentice Hall, 1989.

[24] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, Parts I and II*, Information and Computation **100** (1992), pp. 1–77.

[25] Moller, F., "The Edinburgh Concurrency Workbench (Version 6.1)," Department of Computer Science, University of Edinburgh (1992).

[26] Noll, T., *On coherence properties in term rewriting models of concurrency*, in: *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, LNCS **1664** (1999), pp. 478–493.

[27] Verdejo, A. and N. Marti-Oliet, *Executing and verifying ccs in maude*, Technical Report TR–SIP–99–00, Universidad Complutense de Madrid, Spain (2000).

[28] Viry, P., *Rewriting: An effective model of concurrency*, in: *Proceedings of PARLE '94 – Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science **817** (1994), pp. 648–660.

[29] Viry, P., *Rewriting modulo a rewrite system*, Technical Report TR-95-20, Dipartimento di Informatica, Universita di Pisa (1995).
URL `ftp://ftp.di.unipi.it/pub/techreports/TR-95-20.ps.Z`